Procedia Computer Science

Volume 51, 2015, Pages 1453–1462

ICCS 2015 International Conference On Computational Science

Retargeting of the Open Community Runtime to Intel Xeon Phi

Jiri Dokulil and Siegfried Benkner

Research Group Scientific Computing, University of Vienna, Austria
(jiri.dokulil,siegfried.benkner)@univie.ac.at

Abstract

The Open Community Runtime (OCR) is a recent effort in the search for a runtime for extreme scale parallel systems. OCR relies on the concept of a dynamically generated task graph to express the parallelism of a program. Rather than being directly used for application development, the main purpose of OCR is to become a low-level runtime for higher-level programming models and tools. Since manycore architectures like the Intel Xeon Phi are likely to play a major role in future high performance systems, we have implemented the OCR API for shared-memory machines, including the Xeon Phi. We have also implemented two benchmark applications and performed experiments to investigate the viability of the OCR as a runtime for manycores. Our initial experiments and a comparison with OpenMP indicate that OCR can be an efficient runtime system for current and emerging manycore systems.

Keywords: Open Community Runtime, Intel Xeon Phi, runtime systems, programming models

1 Introduction

The extreme scale and increased performance variability of future high performance computing systems pose many new challenges to parallel programming models and runtime systems. The Open Community Runtime (OCR, [9]) is a recent effort for a runtime system for extreme scale parallel systems. Key features of the OCR are the event-driven, asynchronous task-based approach for expressing parallelism and its support for fault-tolerance. Similar to other task-based systems, the OCR does not give the developer direct control of what is being executed by each thread or process at any time, but rather lets the developer express the parallelism available in an application by creating a large number of tasks, which perform parts of the computation. Synchronization is expressed by specifying dependencies between the tasks.

Achieving good scalability in such task-based system is not trivial, since the task scheduler may incur significant overhead. Solving these issues usually requires significant effort on two fronts: on the design and implementation of the task runtime (API, scheduler, memory management, etc.) and also a major effort by the application developer, or, a higher-level programming model, which creates the tasks to be executed by the runtime.

Selection and peer-review under responsibility of the Scientific Programme Committee of ICCS 2015

doi:10.1016/j.procs.2015.05.335

All of these issues are further complicated when manycore systems like the Intel Xeon Phi coprocessor come into play. The high number of threads significantly increases the cost of cache-line sharing, which may significantly limit scalability of both locks and lock-free structures used to implement the task scheduler.

We have implemented a scheduler which complies with the OCR API, but only runs on a single shared-memory machine, which is either a traditional SMP machine or the Xeon Phi coprocessor. Using two benchmark applications, our goal is to investigate whether the OCR programming model is suitable for the manycore architecture of the Xeon Phi. Since a significant portion of the scheduling has to be done locally to have any chance of achieving good efficiency on an Exascale machine, this can provide us with a basic idea about the potential of OCR for future systems with manycore accelerators. Based on our experiments which compare the OCR implementation to an OpenMP implementation (the OpenMP runtime was highly tuned by Intel for the Xeon Phi), we believe that OCR has the potential to perform well on the Xeon Phi. The performance of well-designed applications realized on top of OCR is close to what can be achieved with comparable OpenMP implementations.

The rest of the paper is organized as follows. Section 2 introduces the Open Community Runtime and outlines our implementation of the OCR API. Section 3 describes the applications which were used in our experiments. Section 4 presents experimental results. Section 5 discusses related work followed by a conclusion in Section 6.

2 Open Community Runtime

The Open Community Runtime is a work-in-progress effort to create a low-level, task-based runtime for extreme-scale parallel systems, with fault-tolerance support. The basic idea of the OCR is to express the computation using tasks referred to as *event driven tasks* (EDTs) and task dependencies, while organizing all data in *data blocks* which are managed by the OCR runtime. The task approach should help the computation dynamically adapt to large scale systems with high variability of performance. With the data blocks and their explicit linking to tasks through dependencies, the runtime is able to move a task to a different node or restart the task on a different node (with the original data) should a node fail.

2.1 Overview of the OCR concepts

The execution of tasks is controlled by dependencies with the help of *events*. A dependency connects a *post-slot* of an event to a *pre-slot* of a task or another event. After the dependency is set up, it can be satisfied, possibly also providing a data block. It is also possible to omit the first event and directly satisfy a dependency by providing a data block. The events can be either freely created as standalone entities or they can be created as part of task creation. In the second case, the event is triggered once the task finishes. The task may return a data block, which is then used to satisfy the event's dependency. If the event is used in another dependency, it will forward the data block along that dependency. It is possible to mark a task as a `finish` tasks, in which case the associated event is triggered once the task and all its children (child tasks) created directly or indirectly by that task have finished. Figure 1 shows basic examples of dependencies.

A task can only access data in a data block if it created the data block or the data block was passed to the task along one of its dependencies. As a result, the runtime is aware of all the data objects a task can possibly access. There are four different access modes, which control the way data blocks can be concurrently accessed by multiple tasks. An access mode is

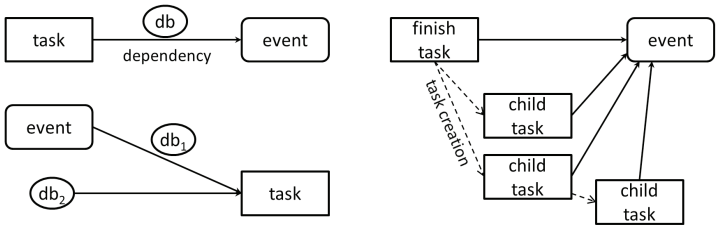

Figure 1: Task dependencies in OCR. On the top left, the post-slot of a task is connected to an event, which is triggered by completion of the task. The data block (denoted "db") can be returned by the task and passed to the event. On the bottom left, two possible dependencies connected to a task pre-slot are show – an event or a data block (which directly satisfies the pre-slot). On the right, the structure of a `finish` task is shown. The event is only triggered after the task and all child tasks have finished.

specified when a dependency is set up and it is valid for the data block which is passed along that dependency. The access modes are: read only (RO; coherent), non-coherent read (NCR; data may be concurrently modified by other tasks), intent to write (ITW; non-coherent access, possibly writing to the data), and exclusive write (EW; coherent, needs to be the only writing task). The OCR runtime may create multiple copies of the data to satisfy the access modes, for example to allow one task EW access to the original, while multiple tasks have RO and NCR access to the copy. The non-coherent modes provide some basic guarantees regarding reads and writes, but it is mainly the programmer's responsibility to maintain consistency of the data in case of concurrent access.

2.2 Simple OCR application example

Consider the following pseudo-code example, where the initial task creates a data block and two child tasks process the data block in parallel:

```
db initial_task() {
  (t1,e1) = create_task(process,1);//Create task t1 and event e1. Task t1 has
      one pre-slot, executes the function process, and triggers event e1 when
      finished.
  (t2,e2) = create_task(process,1);//The pair (t2,e2) is analogous to (t1,e1)
  (t_final,e_final) = create_task(finalize,3);//Three pre-slots, runs finalize.
  db data = create_data();//Create a new data block.
  add_dependency(data,t_final,EW);//Bind the data block to a pre-slot of t_final
  add_dependency(e1,t_final,EW);//Bind post-slot of e1 to a pre-slot of t_final;
  add_dependency(e2,t_final,EW);//the mode is irrelevant as no data is passed.
  add_dependency(data,t1,MODE1); add_dependency(data,t2,MODE2);
  return NULL;
}
db process(db arg) { //arg contains the data block
  work(arg);
  return NULL;
}
db finalize(db arg, db arg1, db arg2) { //arg1 and arg2 will be NULL
  print_result(arg);
  return NULL;
}
```

The corresponding dependence graph is shown in Figure 2. Assuming that the degree of parallelism in the system is at least two, the execution of tasks `t1` and `t2` depends on `MODE1`

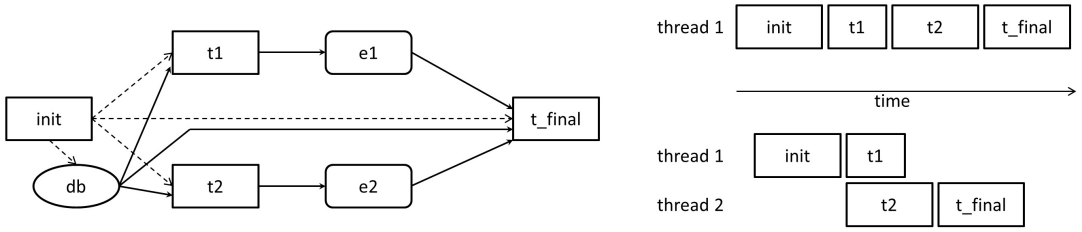

Figure 2: Simple example of a dependency graph (left) and two possible execution timelines (right). On the top, either only a single thread is available or t1 and t2 cannot run in parallel due to data access modes. On the bottom, the tasks can be executed in parallel.

and `MODE2`. For example, if both modes are EW (exclusive write), the execution of the tasks needs to be serialized. If the modes are any combination of NCR and ITW, the tasks may be executed concurrently using a single copy of the data. If `MODE1` is RO and `MODE2` is ITW or EW, the tasks can be executed in parallel, if the runtime makes a copy of the data for `t1`. If `MODE2` is RO or NCR, both tasks can run in parallel over the same data. Note that `t1` and `t2` can start executing right after the last two `add_dependency` calls, since all of their dependencies have been satisfied. There is no need to wait for the initial task to finish. Figure 2 also shows the possible execution timelines.

2.3 Shared-memory OCR implementation on top of TBB

In September 2014, the OCR working group released version 0.9 of the interface specification. Combined with the fact that the OCR is deployed as a library, this makes it easy to run the same application with multiple implementations of the OCR. There is also an ongoing development effort on the implementation for distributed-memory systems, with planned release in 2015.

We have created an OCR implementation which complies to the 0.9 specification, with the exception of some of the experimental and extension APIs. It supports only shared memory multi-/many-core systems and is built on top of the Intel Threading Building Blocks (TBB, [7]) library, most notably the tasks scheduler provided as part of the TBB. Our runtime provides the necessary task dependency management, including access control to data objects. The TBB tasks may also have dependencies, but they are much simpler compared to the OCR making an extra dependency layer a necessity. Basically, the TBB uses a single atomic counter per task to track the number of unsatisfied dependencies and each task has a single pointer to a task that depends on it to decrement the counter and spawn the dependent task if it reaches zero.

OCR allows the runtime to maintain several copies of a data block (e.g., on multiple nodes) and handle reads and writes based on the access mode requested by the application and the OCR memory consistency model. Our implementation always maintains just one copy of any data block and consistency is achieved through locking. Before a task can execute, it must acquire locks on all data objects used in the task. The locks are acquired according to a global ordering to prevent deadlocks. However, these locks are never waited on by any thread of the application. If a lock is needed to run a task, but it cannot be acquired at the moment, the task is added to a queue of tasks waiting for the particular data block. When the data block is unlocked by the owner, it is the owner's responsibility to process the queue of waiting tasks.

Once all pre-slots (incoming dependencies in the dependency graph) have been satisfied and all used data blocks locked, the task is ready to execute. At this point, the task is submitted to the TBB task scheduler, which will execute the task using a pool of worker threads. Once a

task finishes, the event associated with the finalization of the task is satisfied. The satisfaction signal may be propagated further through the dependency graph, based on the OCR rules. All held data objects are released, which may cause other tasks to become ready. All of this is handled in the task epilogue, forming an event-driven dependency tracking system, where no threads are being suspended while waiting for data to become available. The threads only hold fine grained locks which are used to control access to the structures maintained by the runtime and are only held for a brief period, when the structure is being updated.

An important consequence of this approach is the fact that there is no central coordinator or other entity that would imply the use of a highly contested, global lock. This is important for manycore systems, like the Xeon Phi, where such a lock would become a serious performance bottleneck, not only due to serialization of execution but also due to high cost of maintaining such lock through cache-coherency mechanisms.

We believe this approach could also be extended to a distributed environment, but such implementation is beyond the scope of this early experimental study. Our aim is to discover, whether it is possible to implement the OCR interface in such a way that it can efficiently use a manycore machine like the Intel Xeon Phi.

3 Applications

For the experimental evaluation, we have selected two relatively simple simulation applications that represent both highly regular (Seismic) and irregular (SPH) applications. We have implemented two versions of each application, one using OpenMP and the other using the OCR API. All variants were hand-tuned for the experimental machine to give a fair comparison.

3.1 SPH

The smoothed particle hydrodynamics (SPH, [5]) application is an n-body simulation of interacting particles. It comprises a time-step loop that executes in each time-step the same algorithm on the data. For our purposes, the most important property of the SPH is the fact that the particles only interact with other particles that are closer than a specified radius. As a result, the amount of processing required to update each particle depends on the number of neighbors and thus will be different for each particle. The algorithms works in three phases. First, density at the position of each particle is computed, based on the mass of the neighboring particles. Second, pressure forces between neighbors are computed to obtain the overall force and acceleration for each particle. Finally, speed and position of particles gets updated. The first two steps operate as two nested loops. For each particle (the outer loop), all neighbors are processed (the inner loop). The last step is a loop over all particles. In all phases, computation of values of one point is independent on other points. Only values from previous phases are used, no updates have to be propagated within one phase. Since the last phase updates particle positions, the neighbors of a particle change arbitrarily between iterations.

The OpenMP implementation (SPH-OpenMP) uses three parallel `for` loops, which correspond to the three phases of the algorithm. Since in the first two phases the time required to perform single loop iteration (process a single particle) varies from particle to particle, the `guided` scheduling strategy is used for these loops. Both static and dynamic strategies turned out to be significantly less efficient in our experiments. The last loop runs fastest with the default (static) scheduling strategy, since each iteration performs the same work.

The OCR variant (SPH-OCR) uses a single `finish` task to encapsulate each phase. These tasks wait for all of their children to finish, before the event corresponding to the task is

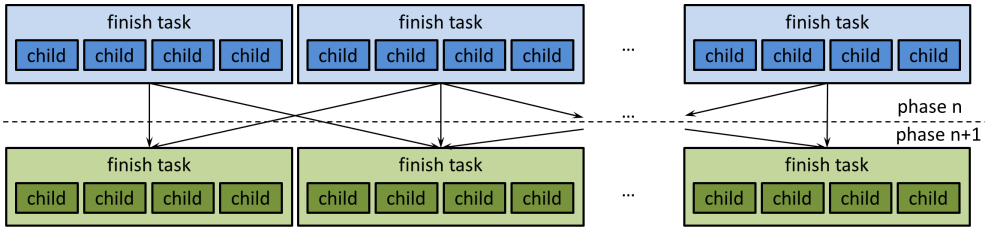

Figure 3: Task dependencies between two phases in the seismic application. The colors match the colors used in Figures 5 and 6.

triggered. The `finish` task creates many tasks, each of which processes several particles. As a result, the execution strongly resembles the fork-join model used in OpenMP. The exact number of tasks and particles processed depends on the actual number of particles, but in all of the experiments the number of tasks is around one thousand. The second phase task has a dependency on the first phase task and therefore there is effectively a barrier between the phases. The third phase follows the same schema. All points are stored in a single large data block, which would make it difficult to execute in a distributed environment, but as a result the whole application was relatively easy to implement. For a developer experienced with the OCR, it would not be significantly more difficult than the OpenMP version. However, it is more verbose, with the source code being around 2.4 times larger.

3.2 Seismic

The seismic application was inspired by the seismic example distributed as part of the TBB library. It simulates propagation of seismic waves through 2D terrain. There are several properties associated to each grid point (stress, velocity, dampening, ...). Like SPH, Seismic runs in several iterations and each iteration comprises three phases. First, the initial seismic pulse is updated, which involves updating velocity of a single point. Second, horizontal and vertical stress is updated for each grid point based on values of properties (other than the stresses) of the point and its neighbors to the right and below. Finally, the seismic wave velocity is updated for each point based on properties (not including the velocity) of the point and its neighbors to the left and above. Like with SPH, there are no dependencies within a phase, just between phases. Unlike SPH, these dependencies are limited only to dependencies between neighbors.

In both variants (OCR and OpenMP), the parallelization is performed by processing rows in each phase independently. This entails two parallel `for` loops in the OpenMP variant. Since all rows require the same work to be done, static loop scheduling is the best option for OpenMP. For the OCR variant, we have used a much more complex (and potentially more efficient) strategy. The data is split into several blocks, each containing the same number of rows (except, possibly, for the last one). These blocks could be distributed to different compute nodes that do not have shared memory (not used in our setup), requiring much less data movement, as only the bordering blocks would have to be synchronized, if the distribution is done in a well-designed way. To provide a reasonable trade-off between the number of data blocks and the available parallelism, each block is processed in parallel by multiple tasks, similar to the SPH-OCR application. However, the dependencies between phases are done in a more fine-grained fashion than the barrier used in the SPH-OCR. A single block of rows is handled by a `finish` task, rather than the whole phase. The `finish` task creates child tasks to process the rows in even smaller blocks. The inter-phase dependencies are set up between these blocks, allowing some

		SPH	SPH-2	Seismic	Seismic-2
Host	OCR time	0.760 (0.016)	25.470 (0.538)	0.046 (0.002)	52.394 (1.126)
	OpenMP time	0.842 (0.030)	25.577 (0.523)	0.158 (0.063)	57.035 (3.871)
	speedup	1.107	1.004	3.462	1.089
Xeon Phi	OCR time	1.955 (0.047)	4.408 (0.049)	0.528 (0.036)	20.106 (0.189)
	OpenMP time	1.843 (0.010)	3.484 (0.006)	0.280 (0.005)	19.985 (0.071)
	speedup	0.942	0.790	0.529	0.994

Table 1: Results of the experiments. The values are averages from 10 executions. The times are given in seconds with standard deviation in parentheses. The speedup is the speedup of the OCR version over the OpenMP equivalent.

task of the subsequent phase to start before all tasks of the current phase finish. An example of the dependencies is shown in Figure 3. Note that the rightmost task in phase $n+1$ does not depend on the leftmost task from phase n. In our implementation, there are 8 of the `finish` tasks and each of them has 80 children. This organization of execution no longer follows the fork-join model and resembles the task-graph techniques used in the TBB.

While the complexity of the OpenMP implementation of the seismic application is comparable to the SPH-OpenMP, the OCR variant is significantly more complicated. However, the more complex design is necessary to allow efficient data distribution and efficient execution on a manycore machine at the same time.

4 Experiments

We have performed several experiments on a host machine with two Xeon X5680 CPUs (3.33GHz, 6 cores, 12MB cache) and Intel Xeon Phi 7120P coprocessor (1.238 GHz, 61 cores, 16GB RAM). Note that in offloading mode, one core is reserved for the operating system, but the experiments were executed in native mode, leaving all 61 cores for the application. With four hardware threads per core, we have up to 244 worker threads. Each experiment was repeated 10 times and the displayed values are the averages.

Table 1 summarizes the results of the experiments. Two variants of each application have been used in experiments. The first variant is the default one, while the second (SPH-2 and Seismic-2) have been modified to make them more computationally intensive. This was done to better explore the overhead of the applications. The results on the host are close to expected, showing some speed advantage of the OCR, most likely due to better efficiency of the TBB thread pool compared to the fork-join model of OpenMP. This is especially apparent with the Seismic application, where the execution time is so small that it is dominated by the overhead. This also explains the improved performance of the SPH application. We can see a significant difference between SPH-2 and Seismic-2. While both the OCR and OpenMP variant of SPH-2 perform nearly the same, the OCR variant of Seismic-2 is noticeably faster than the OpenMP variant. This is probably due to better cache locality of the task-based solution and elimination of the barrier between the phases.

To give some idea of thread utilization and the way the tasks executed, Figure 4 shows a trace of task execution of the SPH on the host. The trace data was gathered using our custom, extremely light-weight collection mechanism, designed specifically for the large number of threads and tasks present in our experiments. Visualization is also done by a custom in-house tool. Figure 5 provides the same visualization for the Seismic application, also on the host. We

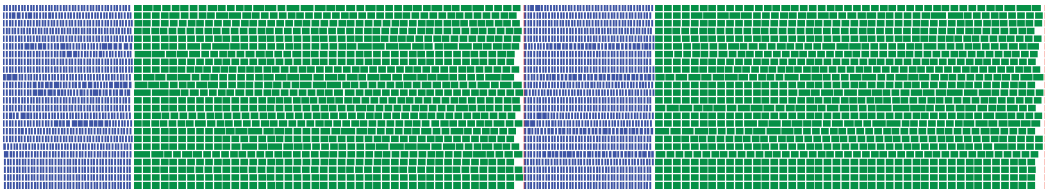

Figure 4: Visualization of tasks used in SPH to execute two iterations of the simulation on the host. Worker tasks of the first phase are shown in blue, the second phase is shown in green. The phased execution is clearly visible. Notice the variable task length, inherent to the SPH.

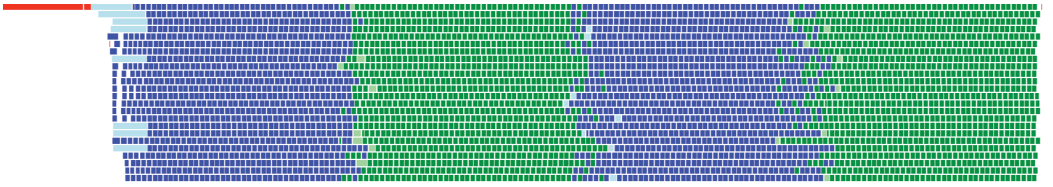

Figure 5: Visualization of tasks used in Seismic (two iterations) on the host. Worker tasks of the second phase (the firth phase is trivial) are shown in blue, third phase is green. The parent (`finish`) tasks are shown in lighter colors. The absence of a barrier between the phases, which allows the task from different phases to mix, can clearly be seen.

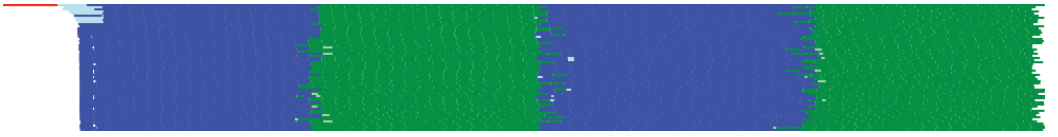

Figure 6: Visualization of tasks used in Seismic on the Xeon Phi.

have the same traces available for the Xeon Phi as well, but the high number of threads make presenting them in the paper impractical. They would paint an analogous picture, as you can see from Figure 6, which shows the Seismic executed on the Xeon Phi, but limited to just 60 threads. The figures follow the layout used in Figure 2, with rows representing the worker threads, each task being represented by a colored rectangle, and the x-axis showing wall-clock time. As you can see, the thread utilization is good – there are very few white spaces once the computation starts. The SPH example in Figure 4 demonstrates the variable task duration, which is a property of the SPH algorithm, where the number of operations necessary to process a particle depends on the number of close neighbors of the particle.

On the Xeon Phi, the measured performance is more interesting. With the fast application variants, the performance of both OCR and OpenMP variants is slower than on the host. This is a result of much larger overhead required to manage much larger number of threads, combined with the fact that the cores on the Xeon Phi are much slower (when compared one-on-one) than on the host. Also, it is more difficult to supply data to such a high number of cores with limited amount of local cache. The OpenMP runtime has been significantly tuned to perform well on the Xeon Phi, whereas the runtime (the task scheduler) of TBB does not scale that well to such high number of threads. The SPH-OCR is an example of an application which was not optimized for a manycore system and as a result, in less-than-perfect setups,

the performance can drop to around 80% of the OpenMP variant. On the other hand, Seismic-OCR was implemented more carefully for these systems, and as a result, the performance of the OCR variant is close to the performance of the OpenMP variant for heavier workloads. With a further modification of the application (not shown in the table), the OCR runs 2.608 seconds and OpenMP 2.332, so the OCR already provides almost 90% of the performance of OpenMP.

It is important to recall that our goal is not to prove that OCR can outperform other programming approaches. Rather we want to assess whether OCR can provide comparable performance in relevant scenarios. Also, the code was not specifically optimized to run on the Xeon Phi, so our results should not be used to evaluate the performance of the Xeon Phi coprocessor, since the code does not yet use the vector units of the Phi to their full potential.

5 Related work

There are several other runtime systems with objectives similar to the OCR. The Nanos++ runtime is used in OmpSs [3] and it relies on GASNet [2] for communication. Unlike the OCR, the Nanos++ runtime has not been designed to be a standard runtime. The OCR data access modes have no direct equivalent in Nanos++, where task-task dependencies would have to be used to synchronize access to data by multiple tasks.

The APGAS (Asynchronous Partitioned Global Address Space) programming model used in the X10 programming language [8] also provides task-based parallelism for distributed-memory applications. However, like other PGAS or MPI-based approaches, the developer is in direct control of data movement and the execution, compared to behind-the-scenes task migration possible in the OCR. Habanero-Java (and later also Habanero-C) were derived from X10 [4].

The Charm++ system [6] uses “chares” rather than tasks to express parallel computation. Chares are objects which can communicate (send messages) with each other and possibly execute a long sequence of interleaved computation and communication, while tasks are pieces of work that should be executed once and that do not communicate with other running tasks, and are not bound to a particular data (block). So, rather than sending a message from a running chare, the task-based solution would be to (once the data is ready to be sent) create a ”send message” task which has access to both sender and recipient.

StarPU [1] is an asynchronous task-based runtime system which has been mainly developed for heterogeneous parallel systems, e.g. CPU/GPU systems. StarPU enables the user to provide different implementation variants (i.e. codelets) for tasks and automates selection and scheduling of proper implementation variants at runtime. It also takes care of data offloading and memory management. Unlike TBB, StarPU provides support for different task scheduling strategies, some of which consider performance models or historical data.

6 Conclusion

Based on the experiments with our implementation of the OCR, we believe it may be a viable tool for development of applications for manycore systems like the Xeon Phi. However, the application has to be carefully designed and implemented or a suitable higher-level programming model must be employed. It is crucial to create a sufficient number of tasks (and set up correct dependencies) in order to utilize the high number of threads on the Xeon Phi efficiently. This is similar to the way the TBB is designed.

The TBB task scheduler has a small, well defined API and it executes the task very efficiently, but using a straightforward and predictable scheduling strategy. This puts the burden of efficient

work allocation in great part on the user of the task scheduler, but also provides the tools to do so. By creating the tasks in a certain way and order, it is possible to either enforce the execution of a task on a certain thread or to significantly improve the chances of a task being executed on the right thread at the best time. This allows the user to have some degree of control over the scheduler and for instance improve cache locality of the computation, even though tasks can be migrated between threads by the task scheduler and the victims for task stealing are selected at random. The algorithms provided by the TBB library are one example of such higher-level approach. For example, the `parallel_for` algorithm is implemented in a very sophisticated way, to automatically control the granularity of the tasks. It also checks which tasks have been stolen and uses the information to improve cache utilization. While this approach provides no guarantees about the scheduling efficiency, it works extremely well on average. Even though the task scheduler and the parallel algorithms form a very sophisticated system, they actually use very few instructions to do their work and generate very little overhead.

We believe that the OCR could play a similar role like the TBB scheduler, but for extreme-scale parallel systems. It will probably be necessary to give the higher levels more control over the way the tasks are moved between the nodes, since the penalty for a bad decision is much higher than the one the TBB has to pay in such a case (i.e., some cache misses). The OCR already provides affinity control as an experimental API, which should be sufficient for this role. We plan to explore these research directions in the future.

Acknowledgment

This work was partially supported by the European Commission's FP7, grant no. 288038, AutoTune.

References

[1] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[2] Dan Bonachea. Gasnet specification, v1.1. Technical report, Berkeley, CA, USA, 2002.

[3] J. Bueno, J. Planas, A. Duran, R.M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive programming of gpu clusters with ompss. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 557–568, May 2012.

[4] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[5] R. A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics - theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389, 1977.

[6] Laxmikant V. Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. In *Proc. of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 91–108, New York, USA, 1993. ACM.

[7] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(04):309–322, November 2007.

[8] Olivier Tardieu et al. X10 and apgas at petascale. In *Proc. of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 53–66, New York, USA, 2014. ACM.

[9] OCR working group. The Open Community Runtime Interface, September 2014. `https://xstackwiki.modelado.org/images/1/13/Ocr-v0.9-spec.pdf`.